Final Project Report *ICN, Spring 109*

Group 2 , member: B07901028黃靖傑
B07901122王譽凱
B07901133顏子鈞

Game Server Implementation:

1. Send/receive game information using TCP and UDP socket:
   - Encode/read any useful information into/from packets, specify as different unique name of header.

2. For any interaction during the game, we classified data into 2 type:
   i. Data that **cannot** tolerate latency.
      - Eg. Player position & movement, weapon's action, timer of game
   ii. Data that **can** tolerate some latency.
      - Eg. Progress bar update

   - For type i, we choose to implement best-effort service(**UDP**), as these data is transferred **according to rate of transfer** *(until the game ends)*, UDP is a better choice when transferring immediate information that cannot tolerate latency.
   - For type ii, we choose to transfer data with **TCP**, as any packet-loss will affect the game logic that is on going

3. What in game logic does our server handles:
   i. Spawning new player when a new client connects the IP address of server, and broadcast this information to everyone else.
   ii. Send the remaining game timer to the latest connecting player, and everything else that has already exist in the game scene.
   iii. Broadcast every player's immediate position & action back to every player, to let everyone render the correct scene of the game.
   iv. When the progress bar updates, broadcast the information of the latest progress bar to every player.
   v. When a player disconnects, destroy its instance of game object, close the corresponding socket.
   vi. When times up, disconnect every client from the server, close all socket.

Client Implementation:

1. When connecting to the server IP address:
   - initiate a *3-way handshaking* **TCP** socket, *packet buffer size* is set to 4096 byes. After correctly read any packet that is sent from server, dispose the packet to release memory for better gaming experience.
   - Any information(packet) that is broadcast instantly from the game server is read from the **UDP** port.

2. When client performs any action in the game, such as moving around, gathering resource, or firing shots:
   - Every time client is **giving input** (such as pressing any button), client will be **sending packets to server**, to update all other players' scene.

3. When client receives any packet from server, game application will handle the corresponding game logic, such as rendering other's gun shot/bomb or the direction of the gun, interaction of mining resource, or update the latest progress bar of resource.

4. We guarantee that all the in game utility(that TA has implement) is 100% preserved, in our multiplayer game:

Player movement & position
Red team gather resource's interaction and animation
Red team head's icon when collected resource
Red team update resource bar at lab, all player's progress bar will update at the same time
Render the rocket image correspond to the current progress bar value
Red & Blue team both can put bomb
Blue team can show/ hide gun, and display the direction of the gun pointing.
Blue team can fire gun shot, making resource collected by Red team to disappear
Projectile of the bullet
Blue team can also be hit by the bullet… (friendly fire)
Red/Blue team getting hit by bomb/bullet will display the "stunned" image

5. We do have implement some extra game logic, such as:
   - The cooldown of the bomb/gun fire is set to *2.5 secs*, because it is unplayable for red team if the blue team player can fire gunshot non-stop, thus impossible for red team to gather resource (*that requires 2 secs)*

Network Optimization:

1. We use *Unity.NetworkEngine* to estimate the current RTT of transferring data, every time we update any scene change, the effective action needs to consider the RTT of the network traffic. For example, the timer of the on going game is not accurate when the network traffic is busy, or the bullet projectile will delay.
   - Eg. Game Timer: if the current time remaining is 4:30 (270s), and the current RTT is 0.05s, then server will broadcast the timer as (270 – 0.05/2) = 269.975s to client, so when client received this timer data, the actual time remaining is around 269.975s (not 270s anymore).

2. If the current network traffic is busy, our player movement will seem to be lagging, so we solve this problem using **Interpolation** of data.
   - For example:
     Actual player position per frame:
     (0,0,0) -> (2,0,0) -> (4,0,0) -> (4,2,0) -> (4,4,0)
     But if packet loss happen, the likely scenario of our client:

(0,0,0) -> x -> x -> (4,2,0) -> (4,4,0) *x: packet loss*

We will experience "lagging" on display as our player prefab "teleports" from (0,0,0) to (4,2,0) instantly.

- What we improve is:
  Next position = 0.7*(previous position) + 0.3*(actual position)
  (0,0,0) -> (0,0,0) -> (0,0,0) -> (1.2, 0.6,0) -> (2.04, 1.62,0) -> (2.628, 2.334,0) -> ….. -> (4,4,0)
  and thus having few more frame at the ending, so that the display seems "smoother" for our client.

3. We have do research about the rate of transfer data(packet). If the rate is too high (like *30 packet per second*), it is likely to increase our server's burden thus having packet loss issue. By not effecting our gaming experience, we set the **rate of transferring** = *15 packet/s*, so that we decrease the probability of packet loss under high network stress test, and preserving smooth gameplay for our client.